템플릿(Template) 1

박 종 혁 교수

UCS Lab

Tel: 970-6702

Email: jhpark1@seoultech.ac.kr

템플릿(Template)에 대한 이해와 함수 템플릿

함수 템플릿 소개

- 함수 템플릿
  - 한번의 함수 정의로 서로 다른 자료형에 대해 적용하는 함수
  - 예

```
int abs(int n)                    // 정수형 인수의 abs() 함수
{  return n < 0 ? -n : n; }

double abs(double n)              // double 형 인수의 abs() 함수
{  return n < 0 ? -n : n; }
```

    - 함수의 인수와 반환형의 자료형 만이 다르고 함수 내부의 프로그램 코드는 모두 같음
    - 템플릿 함수를 사용하여 함수 정의는 한번 만 하고 필요 시 자료형에 따라 각각 적용

함수 템플릿 정의

- 함수 템플릿은 template 키워드, typename(또는 class) 템플릿 자료형 인수들를 선언
  - 템플릿의 자료형이 되어 템플릿 함수의 가인수나 반환형을 위한 자료형으로 사용

```
template <typename Type1 [, typename Type 2 .....]>
    반환형 함수명(가인수 리스트)
    {
        ..................
    }
```

- 함수 호출 시에 사용되는 실인수, 반환형의 자료형을 컴파일러가 템플릿 함수의 자료형 인수에 대입하여 특정 함수를 생성하고 호출

함수를 대상으로 템플릿 이해하기

```
int Add(int num1, int num2)
{
    return num1+num2;
}
```

일반함수

```
T Add(T num1, T num2)
{
    return num1+num2;
}
```

템플릿화의 중간 단계

```
template <typename T>
T Add(T num1, T num2)
{
    return num1+num2;
}
```

템플릿화 완료

```
template <typename T>
T Add(T num1, T num2)
{
    return num1+num2;
}
int main(void)
{
    cout<< Add<int>(15, 20)         <<endl;
    cout<< Add<double>(2.9, 3.7)    <<endl;
    cout<< Add<int>(3.2, 3.2)       <<endl;
    cout<< Add<double>(3.14, 2.75) <<endl;
    return 0;
}
```

T를 double로 하여서 만들어진 함수를 호출하면서 2.9와 3.7을 전달하라!

```
double Add(double num1, double num2)
{
    return num1+num2;
}
```

T를 int로 하여서 만들어진 함수를 호출하면서 15와 20을 전달하라!

```
int Add(int num1, int num2)
{
    return num1+num2;
}
```

컴파일러가 생성하는 템플릿 기반의 함수

```
int Add<int>(int num1, int num2)
{
     return num1+num2;
}
```

Add<int>(…) 의 함수호출 문을 처음 컴파일할때 이 함수가 만들어진다.

```
double Add<double>(double num1, double num2)
{
     return num1+num2;
}
```

Add<double>(…) 의 함수호출 문을 처음 컴파일할때 이 함수가 만들어진다.

호출하기가 좀 불편한 건 있네요.

```
template <typename T>
T Add(T num1, T num2)
{
    return num1+num2;
}
```

호출하기 불편하지 않다! 컴파일러가 전달인자의 자료형을 통해서 호출해야 할 함수의 유형을 자동으로 결정해주기 때문이다.

```
int main(void)
{
    cout<< Add(15, 20)      <<endl;
    cout<< Add(2.9, 3.7)    <<endl;
    cout<< Add(3.2, 3.2)    <<endl;
    cout<< Add(3.14, 2.75)  <<endl;
    return 0;
}
```

전달되는 인자를 통해서 컴파일러는 이를 다음과 같이 해석한다.

Add<int>(15, 20);

전달되는 인자를 통해서 컴파일러는 이를 다음과 같이 해석한다.

Add<double> 2.9, 3.7)

TwoTypeAddFunction.cpp

```
#include <iostream>
using namespace std;

template <typename T>
T Add(T num1, T num2)
{
	cout<<"T Add(T num1, T num2)"<<endl;
	return num1+num2;
}

int Add(int num1, int num2)
{
	cout<<"Add(int num1, int num2)"<<endl;
	return num1+num2;
}

double Add(double num1, double num2)
{
	cout<<"Add(double num1, double num2)"<<endl;
	return num1+num2;
}

int main(void)
{
	cout<< Add(5, 7)                                    <<endl;
	cout<< Add(3.7, 7.5)                          <<endl;
	cout<< Add<int>(5, 7)                         <<endl;
	cout<< Add<double>(3.7, 7.5)          <<endl;
	return 0;
}
```

함수 템플릿과 템플릿 함수

```
template <typename T>
T Add(T num1, T num2)
{
    return num1+num2;
}
```

함수의 형태로 정의된 템플릿이기 때문에 함수 템플릿이라 한다. 즉, 이는 템플릿이지 호출이 가능한 형태의 함수가 아니다.

```
int Add<int>(int num1, int num2)
{
    return num1+num2;
}

double Add<double>(double num1, double num2)
{
    return num1+num2;
}
```

이는 템플릿 함수이다. 템플릿을 기반으로 컴파일러에 의해서 생성된 함수이기 때문이다. 즉, 이는 함수이지 템플릿이 아니다.

둘 이상의 형(Type)에 대해 템플릿 선언하기

```
template <class T1, class T2>
void ShowData(double num)
{
    cout<<(T1)num<<", "<<(T2)num<<endl;
}

int main(void)
{
    ShowData<char, int>(65);
    ShowData<char, int>(67);
    ShowData<char, double>(68.9);
    ShowData<short, double>(69.2);
    ShowData<short, double>(70.4);
    return 0;
}
```

이렇듯 콤마를 이용해서 둘 이상의 형에 대해서 템플릿을 선언할 수 있다.

실행결과

```
A, 65
C, 67
D, 68.9
69, 69.2
70, 70.4
```

템플릿의 선언에 있어서 키워드 typename과 class는 같은 의미로 사용된다.

PrimitiveFunctionTemplate.cpp

```
#include <iostream>
using namespace std;

template <class T1, class T2>
void ShowData(double num)
{
    cout<<(T1)num<<", "<<(T2)num<<endl;
}

int main(void)
{
    ShowData<char, int>(65);
    ShowData<char, int>(67);
    ShowData<char, double>(68.9);
    ShowData<short, double>(69.2);
    ShowData<short, double>(70.4);
    return 0;
}
```

함수 템플릿의 특수화(Specialization): 도입

```
template <typename T>
T Max(T a, T b)
{
    return a > b ? a : b ;
}

int main(void)
{
    cout<< Max(11, 15)              <<endl;
    cout<< Max('T', 'Q')            <<endl;
    cout<< Max(3.5, 7.5)            <<endl;
    cout<< Max("Simple", "Best")    <<endl;
    return 0;
}
```

대소비교 함수 템플릿! 큰 값을 반환한다.

정수, 실수 그리고 문자를 대상으로는 Max 함수의 호출의 의미를 갖는다. 그러나 문자열을 대상으로 호출이 되면 의미를 갖지 않는다!

```
const char* Max(const char* a, const char* b)
{
    return strlen(a) > strlen(b) ? a : b ;
}
```

문자열의 길이비교가 목적인 경우 어울리는 함수!

일반적인 상황에서는 Max 템플릿 함수가 호출되고, 문자열이 전달되는 경우에는 문자열의 길이를 비교하는 Max 함수를 호출하게 할 수 없을까? → 함수 템플릿의 특수화 등장 배경

NeedSpecialFunctionTemplate.cpp

```
#include <iostream>
using namespace std;

template <typename T>
T Max(T a, T b)
{
    return a > b ? a : b ;
}

int main(void)
{
    cout<< Max(11, 15)                          <<endl;
    cout<< Max('T', 'Q')                        <<endl;
    cout<< Max(3.5, 7.5)                        <<endl;
    cout<< Max("Simple", "Best")        <<endl;
    return 0;
}
```

함수 템플릿의 특수화(Specialization): 적용

```
template <typename T>
T Max(T a, T b)
{
    return a > b ? a : b ;
}
template <>
char* Max(char* a, char* b)
{
    cout<<"char* Max<char*>(char* a, char* b)"<<endl;
    return strlen(a) > strlen(b) ? a : b ;
}
template <>
const char* Max(const char* a, const char* b)
{
    cout<<"const char* Max<const char*>(const char* a, const char* b)"<<endl;
    return strcmp(a, b) > 0 ? a : b ;
}
```

함수 템플릿 Max를 char * 형에 대해서 특수화

함수 템플릿 Max를 const char * 형에 대해서 특수화

```
int main(void)
{
    cout<< Max(11, 15)              <<endl;
    cout<< Max('T', 'Q')            <<endl;
    cout<< Max(3.5, 7.5)            <<endl;
    cout<< Max("Simple", "Best")    <<endl;
    char str1[]="Simple";
    char str2[]="Best";
    cout<< Max(str1, str2)          <<endl;
    return 0;
}
```

실행결과

```
15
T
7.5
const char* Max<const char*>(const char* a, const char* b)
Simple
char* Max<char*>(char* a, char* b)
Simple
```

SpecialFunctionTemplate.cpp

```
#include <iostream>
#include <cstring>
using namespace std;

template <typename T>
T Max(T a, T b)
{
    return a > b ? a : b ;
}

template <>
char* Max(char* a, char* b)
{
    cout<<"char* Max<char*>(char* a, char*
    b)"<<endl;
    return strlen(a) > strlen(b) ? a : b ;
}

template <>
const char* Max(const char* a, const char* b)
{
    cout<<"const char* Max<const
    char*>(const char* a, const char*
    b)"<<endl;
    return strcmp(a, b) > 0 ? a : b ;
}


int main(void)
{   cout<< Max(11, 15)<<endl;
    cout<< Max('T', 'Q')<<endl;
    cout<< Max(3.5, 7.5)<<endl;
    cout<< Max("Simple", "Best“)<<endl;

    char str1[]="Simple";
    char str2[]="Best";
    cout<< Max(str1, str2)
            <<endl;
```

함수 템플릿의 특수화(Specialization): 비교

```
template <>
char* Max(char* a, char* b)
{ . . . . }

template <>
const char* Max(const char* a, const char* b)
{ . . . . }
```

특수화하는 자료형의 정보가 생략된 형태

```
template <>
char* Max<char *>(char* a, char* b)
{ . . . . }

template <>
const char* Max<const char *>(const char* a, const char* b)
{ . . . . }
```

특수화하는 자료형의 정보를 명시한 형태

클래스 템플릿(Class Template)

클래스 템플릿

- 클래스 템플릿
  - 한번의 클래스 정의로 서로 다른 자료형에 대해 적용하는 클래스

```
class Counter {
    int value;
public:
    Counter(int n)    { value = n; }
    Counter()         { value = 0; }
     int val()        { return value; }
     void operator ++()  { ++value; }
     void operator --()  { --value; }
};
```

```
class Counter {
    double value;
public:
    Counter(double n)    { value = n; }
    Counter()         { value = 0; }
     double val()     { return value; }
     void operator ++()  { ++value; }
     void operator --()  { --value; }
};
```

클래스 템플릿 정의 및 생성

```
template <typename Type1 [, typename Type 2 ……]>
    class 클래스명 {
        ……
    }
```

```
클래스명 <자료형 [,자료형,…]> 객체명;
```

```
template <typename T>
  class Counter {
    T value;
  public:
    Counter(T n)        { value = n; }
    Counter()           { value = 0; }
    T val()             { return value; }
    void operator ++()  { ++value; }
    void operator --()  { --value; }
```

```
Counter <int> oi;
Counter <double> od;
```

클래스 템플릿 사용 예 1

```
#include <iostream>
using std::endl;
using std::cout;

template <typename T>
   class Counter {
          T value;
   public:
          Counter(T n)            { value = n; }
          Counter()               { value = 0; }
          T val()                 { return value; }
          void operator ++()    { ++value; }
          void operator --()    { --value; }
};
int main(void)
{
   Counter <int> icnt;
   Counter <double> dcnt(3.14);
   Counter <char> ccnt('C');

   ++icnt; --dcnt; ++ccnt;
   cout << "++icnt : " << icnt.val() << endl;
   cout << "--dcnt : " << dcnt.val() << endl;
   cout << "++ccnt : " << ccnt.val() << endl;

   return 0;
}
```

```
#include <iostream>
using std::endl;
using std::cout;
template <typename T>
   class Counter {
          T value;
   public:
          Counter(T n);
          Counter()               { value = 0; }
          T val();
          void operator ++();     { ++value; }
          void operator --()    { --value; }
};
template <typename T>
  Counter<T>::Counter(T n)
{   value = n; }
template <typename T>
  T Counter<T>::val()
{  return value; }

int main(void)
{
   Counter <int> icnt;
   Counter <double> dcnt(3.14);
   Counter <char> ccnt('C');

   ++icnt; --dcnt; ++ccnt;
   cout << "++icnt : " << icnt.val() << endl;
   cout << "--dcnt : " << dcnt.val() << endl;
   cout << "++ccnt : " << ccnt.val() << endl;
   return 0;
}
```

클래스 템플릿의 상속

- 클래스 템플릿도 일반 클래스와 같이 상속
  - 베이스 클래스인 클래스 템플릿을 일반 클래스가 파생 클래스가 되어 상속받는 경우
    - 베이스 클래스인 클래스 템플릿으로부터 생성된 클래스를 상속
    - 클래스 템플릿을 생성하지 않고 전체를 상속 받으면 에러
  - 클래스 템플릿인 베이스 클래스로부터 클래스 템플릿인 파생 클래스가 상속받는 경우

```
template <typename T>
    class Base {
      ......
    };
class Derive : public Base<int> {
        ......
}
```

```
template <typename T>
    class Base {
      ......
    };
template <typename U>
    class Derive : public Base<U> {
        ......
}
```

클래스 템플릿의 상속 예

```
#include <iostream>
using std::endl;
using std::cout;

template <typename T>
   class Area {            // 베이스 클래스 템플릿
         T side1, side2;
   public:
         Area(T s1, T s2)              // 생성자
               { side1 = s1; side2 = s2; }
         void getside(T &s1, T &s2)
               { s1 = side1; s2 = side2; }
   };

template <typename U>
   class Rectangle : public Area<U> {
                              // 파생 클래스 템플릿
   public:
         Rectangle(U s1, U s2) : Area<U>(s1,s2)   // 생성자
               { /* no operation */   }
         U getarea(void)
               {
                 U s1, s2;

                 getside(s1,s2);
                 return s1 * s2;
               }
   };
```

```
int main(void)
{
   Rectangle <int>     ir(3,14);
   Rectangle <double> dr(2.5, 7.3);

   cout << "int area    : " << ir.getarea() << endl;
   cout << "double area : " << dr.getarea() << endl;

   return 0;
}
```

```
int area    : 42
double area : 18.25
```

클래스 템플릿인 파생 클래스 Rectangle이 또 다른 클래스 템플릿인 베이스 클래스 Area로부터 상속받는 예이다. Rectangle 파생 클래스 템플릿는 Area 베이스 클래스 템플릿전체를 상속받는다.

클래스 템플릿의 정의

```
class Point
{
private:
    int xpos, ypos;
public:
    Point(int x=0, int y=0) : xpos(x), ypos(y)
    { }
    void ShowPosition() const
    {
        cout<<'['<<xpos<<", "<<ypos<<']'<<endl;
    }
};
```

일반 클래스

```
template <typename T>
class Point
{
private:
    T xpos, ypos;
public:
    Point(T x=0, T y=0) : xpos(x), ypos(y)
    { }
    void ShowPosition() const
    {
        cout<<'['<<xpos<<", "<<ypos<<']'<<endl;
    }
};
```

클래스의 템플릿화

```
int main(void)
{
    Point<int> pos1(3, 4);        // 템플릿 클래스 Point<int> 형 객체 생성
    pos1.ShowPosition();

    Point<double> pos2(2.4, 3.6);      // 템플릿 클래스 Point<double> 형 객체 생성
    pos2.ShowPosition();

    Point<char> pos3('P', 'F');     // 템플릿 클래스 Point<char> 형 객체 생성
    pos3.ShowPosition();
    return 0;
}
```

템플릿 클래스의 객체생성시 자료형의 정보는 생략이 불가능하다!

PointClassTemplate.cpp

```
#include <iostream>
using namespace std;

template <typename T>
class Point
{
private:
	T xpos, ypos;
public:
	Point(T x=0, T y=0) : xpos(x), ypos(y)
	{ }
	void ShowPosition() const
	{
		cout<<'['<<xpos<<", "<<ypos<<']'<<endl;
	}
};

int main(void)
{
	Point<int> pos1(3, 4);
	pos1.ShowPosition();

	Point<double> pos2(2.4, 3.6);
	pos2.ShowPosition();

	Point<char> pos3('P', 'F');
	pos3.ShowPosition();
	return 0;
}
```

클래스 템플릿의 선언과 정의의 분리

```
template <typename T>
class SimpleTemplate
{
public:
    T SimpleFunc(const T& ref);
};
```

함수의 선언

```
template <typename T>
T SimpleTemplate<T>::SimpleFunc(const T& ref)
{
    . . . .
}
```

템플릿 외부의 함수정의

SimpleTemplate :: SimpeFunc(. . . .)

√ SimpleTemplate 클래스의 멤버함수 SimpleFunc를 의미함

SimpleTemplate<T> :: SimpleFunc(. . . .)

√ T에 대해서 템플릿으로 정의된 SimpleTemplate의 멤버함수 SimpleFunc를 의미함

template <typename T>

√ <T>가 들어가면 이 T가 의미하는 바를 항상 설명해야 한다.

헤더파일과 소스파일의 구분

```
#ifndef _ _POINT_TEMPLATE_H_
#define _ _POINT_TEMPLATE_H_

template <typename T>
class Point
{
private:
   T xpos, ypos;
public:
   Point(T x=0, T y=0);
   void ShowPosition() const;
 };
#endif
```

헤더파일

```
#include <iostream>
#include "PointTemplate.h“
using namespace std;
int main(void)
{
    Point<int> pos1(3, 4);
    pos1.ShowPosition();
    Point<double> pos2(2.4, 3.6);
    pos2.ShowPosition();
    Point<char> pos3('P', 'F');
    pos3.ShowPosition();
    return 0;
}
```

소스파일 1

```
#include <iostream>
#include "PointTemplate.h“
using namespace std;

template <typename T>
Point<T>::Point(T x, T y) : xpos(x), ypos(y) {   }

template <typename T>
void Point<T>::ShowPosition() const
{
    cout<<'['<<xpos<<", "<<ypos<<']'<<endl;
}
```

소스파일 2

기준에 의해서 헤더파일과 소스파일을 잘 구분하였다.

그러나 컴파일 오류가 발생한다! 이유는?

파일을 나눌 때에는 고려할 사항이 있습니다.

```
#ifndef _ _POINT_TEMPLATE_H_
#define _ _POINT_TEMPLATE_H_

template <typename T>
class Point
{
private:
    T xpos, ypos;
public:
    Point(T x=0, T y=0);
    void ShowPosition() const;
 };
#endif
```

헤더파일

```
#include <iostream>
#include "PointTemplate.h“
using namespace std;
int main(void)
{
    Point<int> pos1(3, 4);
    pos1.ShowPosition();
    Point<double> pos2(2.4, 3.6);
    pos2.ShowPosition();
    Point<char> pos3('P', 'F');
    pos3.ShowPosition();
    return 0;
}
```

소스파일 1

위의 소스파일을 컴파일 할 때 Point<int>, Point<double>, Point<char> 템플릿 클래스가 만들어져야 한다. 따라서 Point 클래스 템플릿의 정의 부에 대한 정보도 필요로 한다.

```
#include <iostream>
#include "PointTemplate.h"
#include "PointTemplate.cpp"
using namespace std;
int main(void)
{
    . . . .
    return 0;
}
```

해결책 1. 클래스 템플릿의 정의 부를 담고 있는 소스파일을 포함시킨다.

해결책 2. 헤더파일에 클래스 템플릿의 정의 부를 포함시킨다.

PointClassTemplateFuncDef.cpp

```
#include <iostream>
using namespace std;

template <typename T>
class Point
{
private:
	T xpos, ypos;
public:
	Point(T x=0, T y=0);
	void ShowPosition() const;
};

template <typename T>
Point<T>::Point(T x, T y) : xpos(x), ypos(y)
{ }
template <typename T>
void Point<T>::ShowPosition() const
{
	cout<<'['<<xpos<<", "<<ypos<<']'<<endl;
}
int main(void)
{
	Point<int> pos1(3, 4);
	pos1.ShowPosition();

	Point<double> pos2(2.4, 3.6);
	pos2.ShowPosition();

	Point<char> pos3('P', 'F');    // 좌표정보를 문자로 표시하는 상황의 표현
	pos3.ShowPosition();
	return 0;
}
```

PointTemplate.h

```
#ifndef __POINT_TEMPLATE_H_
#define __POINT_TEMPLATE_H_

template <typename T>
class Point
{
private:
    T xpos, ypos;
public:
    Point(T x=0, T y=0);
    void ShowPosition() const;
};

#endif
```

PointTemplate.cpp

```
#include <iostream>
#include "PointTemplate.h"
using namespace std;

template <typename T>
Point<T>::Point(T x, T y) : xpos(x), ypos(y)
{ }

template <typename T>
void Point<T>::ShowPosition() const
{
    cout<<'['<<xpos<<", "<<ypos<<']'<<endl;
}
```

PointMain.cpp

```
#include <iostream>
#include "PointTemplate.h"
// #include "PointTemplate.cpp"
using namespace std;

int main(void)
{
    Point<int> pos1(3, 4);

    pos1.ShowPosition();

    Point<double> pos2(2.4, 3.6);
    pos2.ShowPosition();

    Point<char> pos3('P', 'F');
    pos3.ShowPosition();
    return 0;
```

배열 클래스의 템플릿화

```
class BoundCheckIntArray { . . . . };
class BoundCheckPointArray { . . . . };
class BoundCheckPointPtrArray { . . . . };
```

자료형에 따른 각각의 배열 클래스

```
template <typename T>
class BoundCheckArray
{
private:
    T * arr;
    int arrlen;

    BoundCheckArray(const BoundCheckArray& arr) { }
    BoundCheckArray& operator=(const BoundCheckArray& arr) { }
public:
    BoundCheckArray(int len);
    T& operator[] (int idx);
    T operator[] (int idx) const;
    int GetArrLen() const;
    ~BoundCheckArray();
};
```

위의 배열들을 대체할 수 있는 배열 기반의 클래스 템플릿

지금까지 설명한 내용을 바탕으로 배열 클래스 템플릿을 직접 완성하기 바란다. 이는 지금까지 공부한 내용의 복습 또는 연습에 해당한다.

ArrayTemplate.h

```
#ifndef __ARRAY_TEMPLATE_H_
#define __ARRAY_TEMPLATE_H_

#include <iostream>
#include <cstdlib>
using namespace std;

template <typename T>
class BoundCheckArray
{
private:
	T * arr;
	int arrlen;

	BoundCheckArray(const BoundCheckArray& arr) { }
	BoundCheckArray& operator=(const BoundCheckArray& arr) { }

public:
	BoundCheckArray(int len);
	T& operator[] (int idx);
	T operator[] (int idx) const;
	int GetArrLen() const;
	~BoundCheckArray();
};

template <typename T>
BoundCheckArray<T>::BoundCheckArray(int len) :arrlen(len)
{
	arr=new T[len];
}

template <typename T>
T& BoundCheckArray<T>::operator[] (int idx)
{
	if(idx<0 || idx>=arrlen)
	{
		cout<<"Array index out of bound exception"<<endl;
		exit(1);
	}
	return arr[idx];
}

template <typename T>
T BoundCheckArray<T>::operator[] (int idx) const
{
	if(idx<0 || idx>=arrlen)
	{
		cout<<"Array index out of bound exception"<<endl;
		exit(1);
	}
	return arr[idx];
}

template <typename T>
int BoundCheckArray<T>::GetArrLen() const
{
	return arrlen;
}

template <typename T>
BoundCheckArray<T>::~BoundCheckArray()
{
	delete []arr;
}

#endif
```

Point.h

```
#ifndef __POINT_H_
#define __POINT_H_

#include <iostream>
using namespace std;

class Point
{
private:
    int xpos, ypos;
public:
    Point(int x=0, int y=0);
    friend ostream&
    operator<<(ostream& os, const
    Point& pos);
};

#endif
```

Point.cpp

```
#include <iostream>
#include "Point.h"
using namespace std;

Point::Point(int x, int y) : xpos(x),
    ypos(y) {  }

ostream& operator<<(ostream& os,
    const Point& pos)
{
    os<<'['<<pos.xpos<<",
    "<<pos.ypos<<']'<<endl;
    return os;
}
```

BoundArrayMain.cpp

```
#include <iostream>
#include "ArrayTemplate.h"
#include "Point.h"
using namespace std;

int main(void)
{
    /*** int형 정수 저장 ***/
    BoundCheckArray<int> iarr(5);

    for(int i=0; i<5; i++)
            iarr[i]=(i+1)*11;

    for(int i=0; i<5; i++)
            cout<<iarr[i]<<endl;


    /*** Point 객체 저장 ***/
    BoundCheckArray<Point> oarr(3);
    oarr[0]=Point(3, 4);
    oarr[1]=Point(5, 6);
    oarr[2]=Point(7, 8);

    for(int i=0; i<oarr.GetArrLen(); i++)
            cout<<oarr[i];

    /*** Point 객체의 주소 값 저장 ***/
    typedef Point * POINT_PTR;
    BoundCheckArray<POINT_PTR>
    parr(3);
    parr[0]=new Point(3, 4);
    parr[1]=new Point(5, 6);
    parr[2]=new Point(7, 8);

    for(int i=0; i<parr.GetArrLen(); i++)
            cout<<*(parr[i]);
    delete parr[0];
    delete parr[1];
    delete parr[2];
    return 0;
}
```

Q & A